Lessons learned producing an OGSI compliant Reliable File Transfer Service

William E. Allcock, Argonne National Laboratory
Ravi Madduri, Argonne National Laboratory

Introduction

While GridFTP[1] has become the de facto standard for data movement in Grid applications, there are usage scenarios where it is not ideal. In particular, the ability to treat data movement as a "job" similar to computational jobs is of great benefit in some situations. Specifically, the characteristics desired in this service were to treat the movement of multiple files as a single job, be able to "submit and forget" the transfer, i.e. disconnected operation, and the ability to check status on the job as a whole or detailed status on the files. To meet these requirements we chose to implement an Open Grid Service Infrastructure[2] (OGSI) compliant Grid service that would accept transfer requests and then reliably manage those transfers for the client.

The primary focus of this paper is the design and implementation issues we considered and overcame to produce an OGSI compliant Grid service. It is not a description of the RFT service itself. In the remainder of this document we provide a very brief overview of the functionality and the primary usage scenarios to set the stage for the subsequent discussions. We then address design issues that we thought particularly important and some lessons learned regarding the implementation.

Brief Description of the Reliable File Transfer Service

We envision two primary usage scenarios for the RFT service. One scenario involves a client who wants fine grained, near real time feedback on the progress of the request. Clients that might have these requirements include a Grid meta-scheduler that wants to update its schedule, or some form of status indicator such as a web page or an icon in the Windows tray that allowed the client to monitor ongoing progress. From the perspective of the RFT service this is a classic push scenario. The other scenario might be your typical desktop user who wished to submit a transfer request and then completely disconnect and ignore it until some time in the future when they will connect and request a status. This is the classic pull scenario.

The Reliable File Transfer (RFT) service is, coincidentally enough, intended to transfer files reliably between two GridFTP servers. The client provides a list of source/destination URLs (we are adding support for moving entire directories by just naming the directory) along with optional attributes to control how they are transferred, such as parallelism, TCP buffer size, etc.. The client is returned a Grid Service Handle (GSH), a globally unique name for this service instance, and can use that to check status on the overall transfer request or the individual file transfers. The client can also subscribe for notifications. This is discussed in more detail under the section on Service Data Elements. Two phase commit is used to insure once and only once submission of the request and the request and all subsequent checkpoint data are written to a JDBC compliant database.

We follow the standard factory / instance model described in the OGSI specification. Upon startup the container will start an RFT factory service instance. The RFT factory supports the create_service interface and takes a Transfer_Request that is a SOAP message whose XML schema can be determined by querying the factories service data. An RFT instance is started and passed the transfer request, it is deserialized into an array of Java objects and these are then written to the database, and the private key written to disk for recovery in the event of a service or container crash. At this point, the two-phase commit completes and the CreateService() call returns a GSH to the client. This GSH can then be used to call Start() to begin the transfers. The GridFTP restart and performance markers are written to the database to allow restart. Configuration of the service allows the control of parameters such as maximum concurrency of file transfers. We have not yet added something to the container to control the maximum number of concurrent instances, however it will be relatively easy to do by having the container maintain a count of the active instances.

Design Principles and Decisions

The Open Grid Service Architecture[4] and the Open Grid Service Infrastructure[3] are more than simple programming APIs. They propose a general viewpoint for viewing grid applications and espouse a set of guidelines believed to lead to interoperable, robust, platform and language independent implementations. Such things, however, are always open to broad interpretation. Below we describe our take on the OGSA design principles and the decisions we made while trying to abide by them.

Lifetime Management

One of the primary characteristics that differentiate a Grid service from a web service is that grid services can be transient. Dealing with this transient nature is always a problem in a distributed environment and OGSI provides a method for dealing with this. All services when invoked must provide an initial lifetime, that is a time after which the service may be terminated. The client can send "keep alive" messages to extend this lifetime. In this way, if a failure occurs and no keep alive message is received, the hosting environment may terminate the service and all references to it will be removed from registries. The RFT service observes this model and employees the optional OGSI factory creation method so that each request invokes its own unique instance of an RFT service with a unique Grid Service Handle. While this turned out to be a quite natural and useful methodology, the issue of lifetime management raised some interesting questions. What is the appropriate lifetime of an RFT service instance? Since one of our prime scenarios envisions a user invoking a transfer job that could last for potentially weeks, and wants to run disconnected while only periodically checking on the status, how are keep alives to be handled. Our, perhaps not very good, solution to this problem is simply to request an indefinite lifetime on the service. In an ideal world, one could imagine the service itself, or perhaps some type of "heartbeat monitor" sending the keepalives. As long as the service is moving data and not "hung" one would assume that it should continue to operate. There is a problem with this solution which is that if a client wanted to cancel the job, but could not connect to the service, for instance due to a network outage, the instance would continue to consume physical resources and possibly monetary as well. A client currently does not need to worry about this because they know that it will be terminated at the expiration of the lifetime. It is quite possible that the introduction of OGSI-Agreement[5] will allow for the client to negotiate either type of lifetime management.

Virtualization

One of the primary emphases within OGSI is resource virtualization. We do an excellent job of virtualizing the transport interface. Without affecting the client in anyway, the implementation of the service could be changed; HTTP could be used in lieu of GridFTP, or any other number of changes. This has, in fact, been shown by virtue of the fact that LBL has their own RFT implemented in python, which shares the same interface.

In the ideal OGSA world, everything is represented as a service. The world is not perfect, and likely there will be exceptions to this rule, and certainly we are only just beginning to understand the virtualizations required in certain resource domains. Data access is one of the areas that is currently receiving tremendous attention, but is early in its life cycle. Currently, we pass a list of URLs that specify files and/or directories. In the future, we will likely want to change the interface to RFT so that it would accept a list of source and destination GSHs rather than URLs.

On the source side, the service listed would either represent a single file or a possibly a file system or directory. If the service represents a file, it would have appropriate access interfaces defined on it, quite possibly implemented via a GridFTP server. If the service represented a file system or a directory, the service would implement interfaces that would return an array of GSHs representing the files or possibly other directories or file systems that it contains.

The destination side would operate somewhat differently. This is because no service exists to represent the file since it is not present yet. Essentially, you would need to provide the GSH of the destination file system and possibly a filename (though, in theory, the system could auto generate one). A possible scenario would involve requesting the destination storage service to create a new service to represent the file about to be transferred, and then invoke its "put" interface. The returned handle would provide information about various protocols it can support. The source side would then choose its protocol and invoke its "put" interface and begin the transfer of data to the destination.

Granularity of Virtualization

Virtualization also supports another desirable attribute of OGSA, that of composition of services. Properly virtualized resource can then have their representative services combined in many interesting ways producing higher-level virtualizations, possibly multiple virtualizations of the same thing exposing different functionality for different costs. We had a long debate on how to deal with handling "jobs" of multiple file transfers. One solution to the problem would have involved a single file RFT service and a higher-level job or queuing service. This queuing service could have been the interface presented to the client allowing for the submission of a job of 100 file transfers. The job service would then have invoked the single file RFT service and handled concurrency, notifications, etc.. While we considered this approach, in the end, we chose to combine the functionality in one service. We chose this primarily because the multi-file RFT can be treated as a single file RFT by simply submitting one file and a higher-level service could be invoked. There were also pragmatic reasons for making this decision. In order to implement the directory support with separate services a file system service would have had to be implemented and available on the source and destination hosts. Long term we expect this to be common place, but it is not now and many hosts which have GridFTP servers running do not

have any OGSI services running and it was felt it would be a barrier to adoption of the RFT service if it required the installation of a hosting environment and a separate service.

Service Data Elements

One of the most powerful elements of the OGSI is service data. This is vital for obtaining information about the service during discovery. For instance, some versions of RFT may support more features than others and it is possible for clients to query the service data to determine what interfaces are supported or what version of the service this is. Service data is also valuable as a monitoring tool and we provide a powerful monitoring capability in our implementation of RFT.

We noted in our initial discussion that we envision scenarios requiring both push and pull of status information. The OGSI SDE support is cleanly able to handle both. To support the push scenarios, the OGSI NotificationSource() interface is used to provide fine-grained state change notifications. An interested service may subscribe to these notifications and will receive near real time notification of state change in the transfer (SRC_URL_A to DEST_URL_B has gone from pending to active), the entire job (JOB status has changed from ACTIVE to COMPLETE), or restarts (SRC_URL_A to DEST_URL_B faulted, retrying). These messages were designed to be small to minimize network overhead and were assumed to occur at a moderate frequency on the order of a few per minute per instance.

The other scenario involves disconnected operation with infrequent asynchronous checks of transfer status, i.e. pull model. This is accomplished by using the OGSI FindServiceData() interface. The response to this query is essentially an array of every transfer in the job along with status. This can be a rather large XML blob, but as it is anticipated to happen only rarely, this was considered acceptable.

Implementation Issues

As with any complex system, there are "tricks of the trade" that can make a significant difference in the success of a given implementation. We note some of the key issues that we dealt with during the development and testing of the RFT service.

SOAP Message Processing Issues

As we began stress testing our implementation and working with user communities to get their feedback, we discovered one substantial issue. The processing of the SOAP message to the RFT factory caused two problems. First, the deserialization of the XML into the DOM tree could take a substantial period of time, on the order of 10s of minutes. Currently, you cannot begin to process the request until it has been completely deserialized, so we cannot start transfers while this is in progress. Second, there was an upper limit of approximately 500 entries before memory ran out. This problem has not yet been resolved, but we have several ideas on how to resolve it, some of which are fairly straightforward and will contribute substantially to improving this situation. First, our initial SOAP message format included a full listing of every possible parameter for every transfer. This meant that for every source/destination pair, there were some 11 XML fields to be processed. We are currently reimplementing the interface to allow for the specification of a set of defaults and then allowing for the defaults to be overridden on a transfer-by-transfer basis, if that is desired. This meant an overhead of 9 field message to start, but a reduction to only two fields, source URL and destination URL for each transfer. Further, we are

adding support for specification of a directory. Currently, if the intent is to move a directory containing 1000 files, the client has to generate a SOAP message containing 1000 entries. We are going to allow the client to specify the directory and the RFT service will contact the source host and expand the directory. This will also likely drastically reduce the size of the SOAP request, improving both deserialization speed and scalability. Finally, we are also planning on looking at the container handling of deserialization and DOM generation.

Standardization of the Interface

Reliable File Transfer is an infrastructure level service and can benefit greatly from having its interface standardized. However, we have not yet begun the standardization process. This is primarily due to the knowledge that OGSI-Agreement was in progress and would almost certainly cause the interface to change. Once OGSI-Agreement has stabilized we will develop an appropriate agreement interface for RFT and begin the standardization process. Ideally, this may simply "fall out" of the Data Access and Integration (DAI) activity within GGF.

Language specific data type

Another of the key features of OGSI is language and platform independence. However, certain development practices can prevent this. In particular, employing data types that are not universally supported can mean that a client for another language can not read the WSDL and auto generate the stubs since it has no way to translate the non-standard types. For instance, Java supports a vector data type. However, C does not. So, the WSDL to Java utility could possibly deserialized the message. However, a WSDL to C utility would have no way of doing so. The moral of the story is stick to base data types that are supported in standard WSDL.

The AnyHelper API

A particularly useful API is provided for handling XML blobs. OGSI uses what is called the "doc literal" format of XML messages. This means that the type descriptions may be determined by introspection. Early in the life of OGSI the auto client tools that would generate the stubs to convert the XML to language specific data types failed. However, with the introduction of the AnyHelper API, this now functions well. This is normally hidden from the developer, but any extensibility element read, such as results from a findServiceData call, can be passed to AnyHelper and a language specific data type will be returned. This is incredibly useful and save a tremendous about of development time. Though we do not use it, AnyHelper can also return the DOM representation if that is of interest.

Fault Tolerance

One of the benefits gained from running in a web services hosting environment is improved quality of service. The Globus Toolkit V3.0 provides a hosting environment that provides improved fault tolerance via the use of the provided persistence API. The purpose of the persistence API is to allow the service instance to record critical internal state that it can use if it is restarted. Data that is common to all service instances, such as its GSH are stored automatically. In the event of a container crash or an instance failure, the container will read server-config.wsdd for any data indicating there had been running instances. The instances are restarted and then they are expected to make a call to the persistence API to determine if they are a new instance or a restarted instance. If they do indeed discover persisted state they can use that to pick up operation from its last checkpoint. The choice of what state is written out via the

persistence API can have a significant impact on the performance of the restart of the container and instance. In our case, we simply persist the primary key of the transfer record, which contains all the information about the request and the current state in a JDBC compliant database. We can then use this to query the database to find the last checkpoints and statuses of the transfers that had been in progress and using the GridFTP restart functionality. We could have chosen instead to persist the entire current state, but this would have been a significant impact in terms of performance, due to constantly having to update the persisted state and in terms of startup as this is not a highly optimized and large amounts of data to read in can significantly impact the time to resume operation.

Conclusions

The OGSI framework provides standard solution and semantics for problems that are common to Grid computing. It provides ways of managing stateful, transient services. Standard, but extensible introspection methods are supported, advanced hosting environments can provide higher levels of quality of service, and various useful utilities are provided. We found the framework and the semantics extremely useful for the needs of our service. There were challenges along the way, particularly since we were tracking the evolving specification and often had to redo sections of the code, but in the end we have a robust service that we feel fills a key niche in the Open Grid Service Architecture.

Support

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38; by the National Science Foundation; by the NASA Information Power Grid program; and by IBM.

References

[1] Data Management and Transfer in High Performance Computational Grid Environments. B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, S. Tuecke. *Parallel Computing Journal*, Vol. 28 (5), May 2002, pp. 749-771.

[2] Open Grid Services Infrastructure (OGSI) Version 1.0. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling; Global Grid Forum Draft Recommendation, 6/27/2003.

[3] Grid Services for Distributed System Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke. *Computer*, 35(6), 2002.

[4] The Anatomy of the Grid: Enabling Scalable Virtual Organizations. I. Foster, C. Kesselman, S. Tuecke. *International J. Supercomputer Applications*, 15(3), 2001.
Defines Grid computing and the associated research field, proposes a Grid architecture, and discusses the relationships between Grid technologies and other contemporary technologies.

[5] Agreement-based Grid Service Management (OGSI-Agreement) (Draft 0). K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu. *Global Grid Forum, GRAAP-WG Author Contribution*, 12 June 2003.